



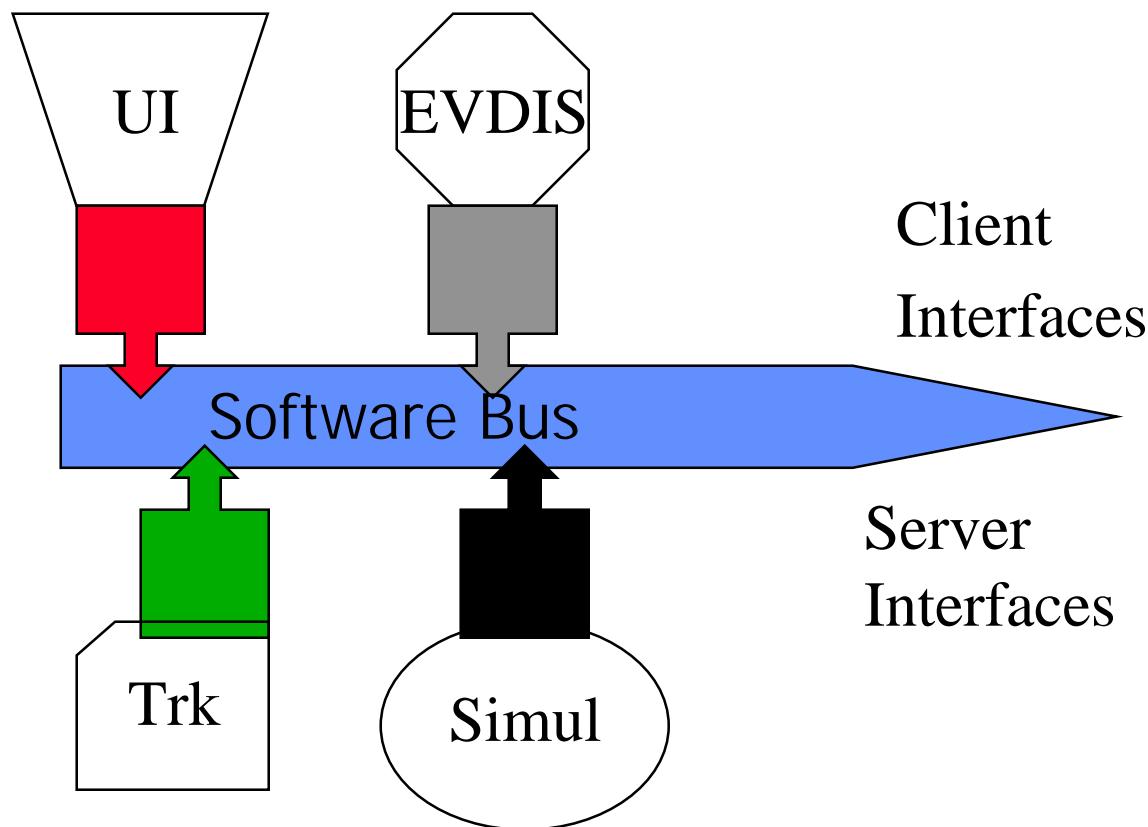
Using a Code Generator for the Atlas Framework

Paolo Calafiura, LBL

Summary Slide

- Motivation: Atlas Framework and Past Experiences
- What we want to generate
- Components of the Generator
- The Description Language: IDL or DDL?
- The Description Language: language neutral?
- A Comparison of Languages
- The Front-End and the API
- Whither Code Generation

A Framework Architecture



Motivation: Atlas Framework

- support/enforce component software design approach:
“program to an interface”, let the generator write the
(skeleton of an) implementation
- support multiple languages, multiple I/O systems
- allow for “graceful retirement” of framework technology
(well, at least non-traumtical)
- provide common module services: registration, serialization
of parameters and status, network communication
(distributed framework)

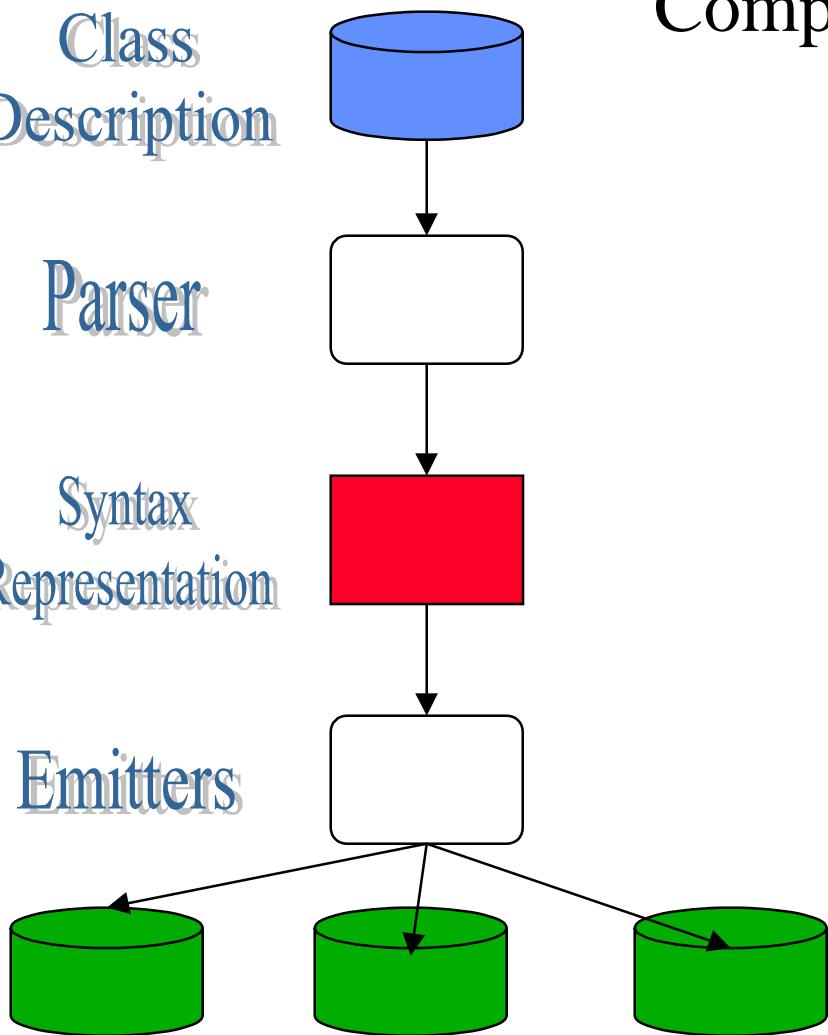
Motivation: past experiences

- Atlas: AGE Geant-3 generator - generate implementation, old technology, new tool in the design phase -ODL based
- BaBar: Fidle - data access from multiple languages - IDL, SUN CFE parser, C++ Eiffel and F90 back-ends
- CDF: DBManager - calibration table I/O from multiple DBMSs - Java based, C++ and Java back-ends for flat files, ORACLE and MSQQL
- Objectivity: interface to target language, I/O, navigation - DDL, custom parser
- ROOT: rootcint - object registration and I/O, run-time method invocation, browsing - C++, cint parser
- STAF: STIC - data table access, analysis module interfaces - IDL, custom parser, C/F77 structures, C++ headers + “glue”

What we want to generate

- ➔ module interfaces:
 - ➔ C++ headers
 - ➔ Java interfaces
 - ➔ scripting language hooks (maybe using SWIG or equivalent)
- ➔ module parameter and status streamers (flat file, memory, DBMS)
- ➔ Software bus adaptors (server skeletons and client stubs)
- ➔ documentation
- ➔ not everything must come out of a single tool

Class Description
Parser
Syntax Representation
Emitters



Components

- A formal description of the interface (e.g CORBA IDL)
- A parser that builds an abstract syntax tree or some other representation
- An API to access the AST in an easy way (e.g. `java.lang.reflect`)
- A set of back-ends (emitters) to produce the code (or whatever is needed)

The Description Language: IDL or DDL?

- for Control, an Interface Description Language is what we want:
 - e.g. CORBA Interface Description Language (IDL)
 - + simple C++ like syntax
 - + industry-standard, used also in HENP (e.g. STAR)
 - + many tools available (Sun CFE parser, JavaCC, ORBs, idldoc,...)
- for event data description, this is probably not enough (associations, templates).
- Do we care?

The Description Language: language neutral?

- If most of the modules are going to be written in C++, why not to use their header file as input to the code generator? (the approach of most commercial ooDMBS)
Good C++ parsers are difficult to find/write and even more to maintain (cint).
- Java comes loaded with tools potentially very useful for the framework.
A description in java would greatly reduce the need of code generation
But `java (as C++) is rich in language-specific idioms, must carefully restrict its usage for the description...
- Is there a language neutral data description language?
 - ODMG ODL(+OIF) very close to Obj DDL, no other known tool. Pretty much dead.
 - OMG XMI (the future exchange format of UML-based CASE tool).
Bleeding edge but interesting (at least one public domain application parse XMI)

A Comparison of Languages

	IDL	ODL/DDL	C++	Java	XMI/MOF
Suitability control	+ interface description	+ superset of IDL	Too complex?	arguments	+
Suitability event	- no persistence, no associations,...	+ Atlas is using it	As IDL	No associations	+
Tools: parsers	Sun CFE, ORBs, JavaCC	Objectivity internal	-cint	JavaCC	Argo Rose?
Tools: API	Interface Repository (ORBs)	Objectivity internal	-Root Meta Library	Language	?
Stability	+	?	+	At least as dict language	?
Lifetime	industry standard, tools to evolve	dying?	+	+	?
Overall	Simple, supported	Much related to Objectivity	Hardest for generator	Easiest for generator	Shot in the dark

The Front-End and the API

- bare-bone parsers (lex+yacc): giant case statement to be filled up
- AST generators (SUN CFE, JavaCC): use the case statement to fill up an abstract syntax tree
We must provide (or find) code to fill out an API a la `java.lang.reflect` to make our life easier.



A code emitter (AST based)

```
//  
//generate table_tabhh_rawstruct -----  
//  
ost.str("");  
skip_to_indent(ost);  
ost << "struct Raw" << endl;  
skip_to_indent(ost);  
ost << "{" << endl;  
increase_indent();  
  
//default constructor  
skip_to_indent(ost);  
ost << "Raw():";  
first = I_TRUE;  
i = new UTL_ScopeActiveIterator(str,  
    UTL_Scope::IK_decls );  
  
while ( !(i->is_done( )) ) {  
    d = i->item( );  
    if ( d->in_main_file( ) ) {  
        if ( AST_Decl::NT_field == d->node_type( ) ) {  
            field = be_field::narrow_from_decl( d );  
            if ( NULL != field ) {  
                if ( I_FALSE == first) { ost<< ", " ; }  
                first = I_FALSE;  
                ost << field->local_name()->get_string()<< "(0)"  
            } else {  
                cerr << "narrow failed on NT_field!" << endl;  
            }  
        }  
        i->next( );  
    }  
    ost << " { }" << endl;  
    delete i;
```

A code emitter (API based)

```
Field f[] = fileClass.getDeclaredFields();
//  
//generate table_tabhh_rawstruct -----  
//
String rawstr="";  
skip_to_indent(rawstr);  
rawstr += "struct Raw {";  
endl(rawstr);  
increase_indent();  
  
//default constructor  
skip_to_indent(rawstr);  
rawstr+="Raw():";  
  
for (i=0; i<f.length; i++) {  
    if (i>0) rawstr+=", ";  
    rawstr += f[i].getName() + "(0)";  
}  
rawstr += " {}";  
endl(rawstr);
```

Which Code Generation?

- Use it to reduce overall work: don't write a parser to generate the interface of 5 classes
- Focus on making end-user life easier: first CDF users can't believe they only have to write a 10-lines dictionary file to put their constants in the calibration database
- Make it part of the standard build process: generate complete, ready to use files
- Focus on the problem at hand: parsing 50% of a language can be enough for 99% of the use cases.